Date Submitted: 09/21/2001

JPL DELIVERABLE DATA SHEET

NASA Office of Safety and Mission Assurance (OSMA) Software Program

Administered by NASA Goddard Space Flight Center (GSFC) - Code IT
Software Independent Verification & Validation Facility
Fairmont, WV 26554

Product Name:	Property-Based Testing, "Tester's Assistant"
Date Completed:	Q4 FY01
Product Version Number:	1.0
Center Initiative Title:	Reducing Software Security Risk Through an Integrated Approach
NASA Program Officer:	Ken McGill
Center Initiative Task Lead:	Dr. David Gilliam, 323-08
Contributors/Authors:	David Gilliam, Ph.D., John Kelly, Ph.D., JPL, and Matt Bishop, Associate Professor UC Davis
Deliverable:	Property-Based Testing, "Tester's Assistant"
Clearance:	UC Davis Property-Based Testing, "Tester's Assistant," has been reviewed and accepted for release by JPL
NASA Project Collaboration:	None
JPL Archive Location:	http://security.jpl.nasa.gov/rssr/

Abstract

The fourth quarter delivery, FY'01 for this RTOP is a Property-Based Testing (PBT), "Tester's Assistant" (TA). The TA tool is to be used to check compiled and pre-compiled code for potential security weaknesses that could be exploited by hackers. The TA Instrumenter, implemented mostly in C++ (with a small part in Java), parsers two types of files: Java and TASPEC. Security properties to be checked are written in TASPEC. The Instrumenter is used in conjunction with the Tester's Assistant Specification (TASpec) execution monitor to verify the security properties of a given program.

Keywords: Software engineering, Security, Property-Based Testing, Vulnerabilities, Execution Monitor, Instrumenter

Deliverable Contents / Attachments

The Property-Based Testing, "Tester's Assistant" (TA) Instrumenter:.
The Instrumenter, implemented mostly in C++ (with a small part in Java), parsers two types of files: Java and TASPEC. As each file is parsed, an AST is built for it. Then three passes are made over each Java AST. During the third pass, instrumented versions of the original Java files are generated. The Instrumenter is used in conjunction with the TASpec execution monitor to verify the security properties of a given program.

The TASpec execution monitor (TEM):
The TASpec execution monitor (TEM) reads the output of a binary that has been instrumented by a TASpec code instrumenter and determines whether or not a specification has been violated. The TEM is implemented in Gnu Prolog.

This version of the TEM takes its input from an ASCII file generated upon exit by the code under test. This is for convenience in testing the TEM and the instrumenter, and can be easily modified to take input in real time.
The TEM deals with two sorts of entities: TASpec statements, which it receives from the code under test, and facts, which reside in the TEM's Prolog database. Significant events in the code under test will trigger the assertion or retraction of facts from the database. Every time the database changes the TEM will check whether or not the current set of facts satisfies all of the rules it has been given via TASpec statements. If a rule is not satisfied the property under test has been violated and the TEM outputs a warning.

TA Instrumenter Documentation
TEM Documentation

PROPERTY BASED TESTING

Tester's Assistant (TA)

TEM Documentation

Table of Contents

TEM Documentation

The TASpec execution monitor (TEM) reads the output of a binary that has been instrumented by a TASpec code instrumenter and determines whether or not a specification has been violated. The TEM is implemented in Gnu Prolog. This version of the TEM takes its input from an ASCII file generated upon exit by the code under test. This is for convenience in testing the TEM and the instrumenter, and can be easily modified to take input in real time.

I. Overview

The TEM deals with two sorts of entities: TASpec statements, which it receives from the code under test, and facts, which reside in the TEM's Prolog database. Significant events in the code under test will trigger the assertion or retraction of facts from the database. Every time the database changes the TEM will check whether or not the current set of facts satisfies all of the rules it has been given via TASpec statements. If a rule is not satisfied the property under test has been violated and the TEM outputs a warning.

II. Prototype Operation

To use the TEM, Gnu Prolog must first be installed on the system and the files specs_checker.pl, em.pl, and booleans.pl must be present in one directory along with the output of the instrumented code. The TEM is then invoked by typing `gprolog` from the directory containing the TEM to start the Gnu Prolog interpreter and typing `[specs_checker] .` at the interpreter prompt to consult the TEM files. Note that the period following `[specs_checker]` *is* required—this is Prolog's way of delineating commands. Analysis is initiated by typing `reader(output).`, where "output" is the name of the instrumenter output file to be checked. The name of the instrumenter output file must **not** contain any capital letters or punctuation. For example, "EM.txt" will not work. But "emtxt" will. (Invalid file names produce a very cryptic error message from the Prolog interpreter. If you get such a message and it refers to "line 2", then you probably have an invalid file name. The "line 2" comes from the input line number of the reader command; line 1 is the specs_checker line.)

If the TEM encounters a policy violation, it will output a warning in the form of:

`Violation of rule:` TASpec invariant causing the violation

If no violation is encountered, the TEM will write `EOF reached` when it has finished analysis. In either case, the Prolog interpreter will then prompt `true?`. Press return to accept the result.

III. Input Semantics

All input to the TEM comes from the instrumented code under test in the form of TASpec statements. Statements are composed of high-level keywords, boolean keywords, facts and variables. Every statement begins with a high-level keyword, to distinguish the seven major types of statements.

The TEM uses Prolog syntax so as to avoid the need for costly parsing within the TEM. This results in some syntactic differences from TASpec.

A. Example

The following is a trace of TEM input received from the execution of an instrumented su program. The policy under which su has been instrumented ensures that authentication is performed before permission are granted. The first three statements represent that policy, the next four were emitted by the instrumented su at appropriate events during execution, and the final **quit** statement simply signals end of execution.

```
exec(check((authenticated(_uid) before permissions_granted(_uid)))).
exec(if(((password_entered(_pwd1) and user_password(_name,_pwd2,_uid)) and
equal(_pwd1,_pwd2)), assert(authenticated(_uid)))).
assert(user_password(atestuser, a1VipHghkqrhQe0x1vOcwuIrS4bNpQE1, a1835)).
assert(password_entered(a1VipHghkqrhQe0x1vOcwuIrS4bNpQE1)).
assert(equal(a1VipHghkqrhQe0x1vOcwuIrS4bNpQE1,
                a1VipHghkqrhQe0x1vOcwuIrS4bNpQE1)).
assert(permissions_granted(a1835)).
quit.
```

The following sections will explain the syntax in which the TEM expects its input.

B. High-Level Keywords

High-level keywords begin each TASpec statement, and may sometimes be nested under other high-level keywords, such as **exec(check(***expr***))** or **forall(***expr***, assert(***fact***))**

- **assert(***fact***)**: An **assert** adds its argument to the Prolog database as a fact and triggers a check of all existing rules against all facts currently in the database.
- **assertonce(***fact***)**: An **assertonce** adds its argument to the Prolog database only until the next check, then removes it again (without triggering a second check).
- **check((***expr***))**: A **check** causes its argument to be checked once against all the facts currently in the database. The double parentheses around the argument are a Prolog quirk.
- **exec(***expr***)**: An **exec** adds its argument to the set of rules to check.

- **if**(*expr*, *exprlist*): For every set of variable instantiations for which an **if**'s first argument is supported by facts in the database, the TEM checks the remaining arguments against the database with those instantiations already made.
- **quit**: A **quit** signals the end of input.
- **retract**(*fact*): A **retract** removes its argument from the Prolog database and fact and triggers a check of all existing rules against all facts currently in the database. Following Prolog syntax, **retract** uses the underscore ("_") as a wildcard variable.

C. Boolean Keywords

Expressions within TASpec statements are built from boolean relationships between facts. Further detail on the peculiarities of boolean logic under TASpec may be found in section IV, B.

- X **and** Y: For some set of variable instantiations, facts exist to satisfy both X and Y.
- X **before** Y: For any set of variable instantiations, facts will exist to satisfy X before facts exist to satisfy Y.
- **eventually** X: When the quit message is received, facts will exist for some set of variable instantiations to satisfy X.
- **not** X: There exists no set of variable instantiations for which facts exist to satisfy X.
- X **or** Y: For some set of variable instantiations, facts exist to satisfy either X or Y.
- X **until** Y: For any set of variable instantiations, for which facts come into existence to satisfy Y, there will have existed—from the moment the until statement was instantiated—facts that satisfy X.

D. Constants and Variables

The atomic elements of any TASpec statement are constants and variables, as used by Prolog. Constants may be atoms or integers. Syntactically, any atom beginning with a capital letter or the underscore character will be interpreted as a variable, while any atom beginning with a lowercase letter (atom) or a digit (integer) will be interpreted as a constant. Both facts and constants may contain (after the first letter) any upper or lowercase letters, digits, and the underscore character. In the example above, the code instrumenter has prepended a lower case "a" to the arguments of every **assert** and **retract** and an underscore to the arguments of every statement that is not an **assert** or **retract**. This forces the database to contain only constants and the rule base to consider only variables, which is a reasonable simplification.

IV. Output Semantics

The TEM's output is fairly simple. If given a bad TASpec statement as input, it will report that fact and ignore the input. If a **quit** input is received before any violation is discovered the TEM reports

successful execution and exits. If a specification violation is discovered, the TEM reports which specification has been violated and exits.

V. Design Decisions

A. Division between Instrumenter and TEM

There are several decisions to be made in drawing the line between the TASpec code instrumenter and the TEM. This implementation puts most of the work on the instrumenter because the TEM (implemented in Prolog) is expected to be the bottleneck in a real time application. The other variation that was considered was to have the TEM read the specification file and handle location specifiers and then have the instrumented code output locations and arguments rather than TASpec statements.

B. Boolean Interpretation

The semantics of the boolean logical operators are well understood, and rarely bear special mention when constructing a language, but TASpec introduces some subtle complications that must be resolved if the language is to be consistent and useful.

The first complication is the fact that the Execution Monitor (EM) operates on logical expressions that may contain variables and may be instantiated in different ways. This creates the potential for an expression that is true for one set of variables but false for another. For example, if the database consists of the assertions

authenticated(bob)
password(bob)
password(alice)

then the expression

authenticated(X) **and** *password(X)*

is true for *bob* but false for *alice*. The expression is also false for *tom*, *dick*, *harry* and the infinite space of other possible instantiations of X. Yet the EM must report either true (secure) or false (insecure). "Secure in some cases" is not acceptable. This example and others like it have required design decisions to disambiguate the EM's response to inputs with more than one reasonable interpretation.

The second complication is the inclusion of the temporal operators **before**, **eventually** and **until** in TASpec. These operators are not so well understood as their logical counterparts, and design decisions were required to give them clear and useful interpretations. The following is a first draft of documentation for these decisions.

For the logical operators, the essential decision is whether to use existential logic, where an expression is considered true if it is true for any possible set of variables, or universal logic, where an expression is considered true only if it is true for all possible sets of variables. Each form of logic produces counterintuitive or unhelpful results in some cases, making a hybrid approach clearly necessary. For example, under an existential approach the expression A **and** B **implies** A < B would be true if any A was less than any B, even if all other A's were greater than all B's—which contradicts the intuitive meaning of **implies**—while under a universal approach the authentication example shown above must always be false simply because X can have an infinite number of possible values while our database certainly cannot hold an infinite number of assertions.

The **and** and **or** operators are therefore interpreted existentially, while **implies** is interpreted universally. **not** is also interpreted universally in that it is taken to mean "There is not any instantiation of variables that will make X true." This set of interpretations seems to produce the most useful and intuitive results, but there remain some tricky inconsistencies. The primary one being the invalidation of DeMorgan's laws. Under TASpec,

A **and** B

means "For some set of variable instantiations, both A and B are true.". While

not((**not** A) **or** (**not** B))

which, by DeMorgan's, should be equivalent, means instead "There is no set of variable instantiations for which either A or B are false." While this is not a major problem, documentation will have to be very clear on the point to avoid confusing specification writers.

Several interpretations for the temporal operators were considered, with the primary decisions being between existential or universal logic and "resetting" or "non-resetting." A resetting interpretation of X **before** Y is "Every time Y becomes false, X will become true before Y becomes true again." The corresponding non-resetting version is "X will become true before Y next becomes true." Either interpretation is fairly intuitive, and cases can be made for the usefulness of both. We have chosen to implement non-resetting operators for the moment, while keeping open the possibility of adding

alternative resetting versions later. The decision between existential and universal logic is simpler on the semantic level, as universal temporal operators are nearly always false and have few practical applications, but implementing non-resetting temporal operators with existential logic has raised some efficiency issues which will require further testing. We believe that we can optimize the implementation and achieve satisfactory performance.